ANALYSIS AND REDUCTION OF LINEAR PROGRAMMING MODELS

V. Messina, University of Milan, Italy
&
S. Moody, Brunel University, UK

G. Mitra (supervisor), Brunel University, UK

October 1992
Revised August 1993
TR/07/93

CONTENTS

1. Introduction

It is well known that 'preserving' a Mathematical Programming problem prior to optimization can dramatically reduce the problem dimension and thus solution time; in some cases it is also possible to detect infeasibility, unboundedness (often caused by formulation errors) or even solve the model prior to applying the simplex method which can often be costly [WILLIAMS, 1990,p35]. Reduction procedures are not only important for the acceleration in solution using the simplex solution, but are also critical for efficient performance of Interior Point Methods (IPM) [LEVKOV,1992].

A method for performing reduction on Mathematical Programming problems, as set out in [BRMIWI, 1975], is to scan the matrix a number of times, eliminating redundant constraints, deriving bounds on shadow prices for singleton columns, removing or tightening variable bounds (and consequently fixing them where possible), replacing singleton rows with simple bounds and detecting unboundedness or infeasibility. This is repeated until the matrix has been passed twice with no reduction.

In the following section of this report, our implementation of this algorithm (with modifications and additions) is explained with reference to the pseudocode for the main program and subroutines provided in Appendix A. In section 3 the data structure we used in implementing the Presolve algorithm is presented with some discussion concerning the communication with FortLP. Section 4 contains results of various problems (including some from netlib) showing the reduction achieved with Presolve. Section 5 discusses the need for post processing in order to reconstruct the formally optimal basis starting from the optimal solution of the reduced problem and proposes additional reduction procedures which could be incorporated into Presolve. The final section contains some concluding remarks and discusses the implications of using Presolve for integer programming (at each stage of branch and bound).

2. Presolve Algorithm

The presolve program described here is based on the reduction algorithm by Brearley, Mitra and Williams [BRMIWI,1975]. For simplification the following primal and dual problems are considered throughout this report:

Primal Problem

$$\begin{aligned} & Max \sum_{j=1}^{n} c_j x_j \\ & subject\ to \sum_{j=1}^{n} a_{ij} x_j \le b_i \qquad i=1,2,\ldots,m \\ & \qquad l_j \le x_j \le u_j \qquad j=1,2,\ldots,n \end{aligned} \tag{2.1}$$

Dual Problem

$$\begin{aligned} & Min \sum_{i=1}^{m} b_i v_i + \sum_{j=1}^{n} u_i y_i - \sum_{j=1}^{n} l_i w_i \\ & Subject\ to \sum_{i=1}^{m} a_{ij} v_i + y_j - w_j = c_j \qquad j=1,2,..n \\ & \qquad v_i,\ y_j,\ w_j \ge 0 \qquad i=1,2,\ldots,m \\ & \qquad\qquad\qquad\quad j=1,2,\ldots,n \end{aligned} \tag{2.2}$$

Although it may seem restrictive to only consider constraints of type "less than or equal to" the subsequent rules are easily adapted for "equality" or "greater than or equal to" constraints [BRMIWI,1975]. Similarly, a minimisation problem may be dealt with by negating the objective row coefficients and maximizing. The variable bounds, l_j and u_j, may be finite or infinite, ie. the variables x_j may be considered as free.

The program consists of five subroutines which perform various types of reduction: namely, variables are fixed at their upper or lower bounds; unboundedness or infeasibility is detected, redundant rows are set free; variable bounds are tightened. The algorithm is recursive with reduction in one pass leading to further reduction in the next (see main program in Appendix A). Figure 1 shows the type of reduction performed by each subroutine.

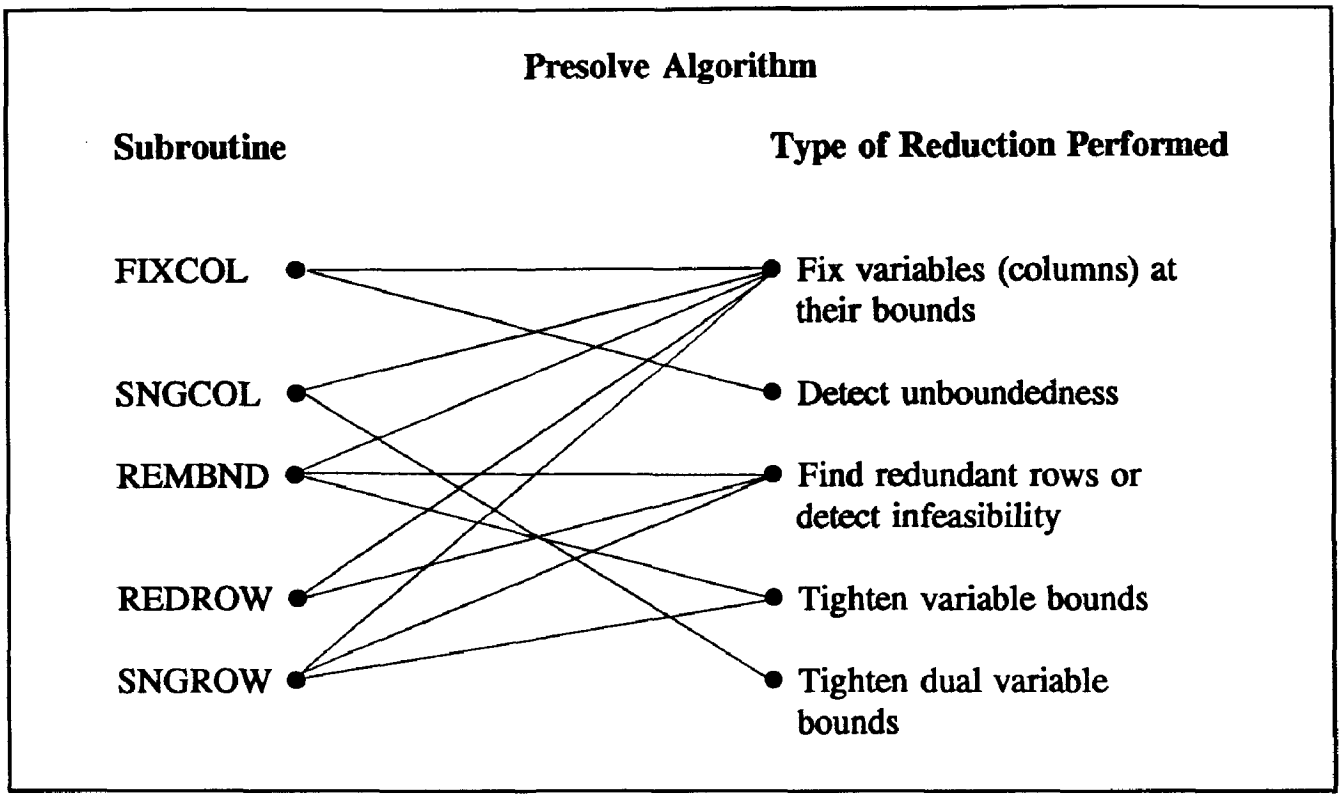


Figure 1 *Presolve Subroutines*

2.1 FIXCOL - fixing variables at their bounds

The subroutine FIXCOL by analysing the upper and lower costs Pj and Qj (defined below) together with the primal cost coefficients, fixes the variables (where appropriate) at their bounds.

Thus for a particular primal variable k, the lower and upper costs, P_k and Q_k are given by

$$P_k \le \sum_{i=1}^{m} a_{ik} v_i \le Q_k \qquad (2.3)$$

where P_k and Q_k are computed from the lower and upper bounds on the dual variables, p_j and q_j, as follows:

$$P_k = \sum_{i \in S_k} a_{ik} p_i + \sum_{i \in T_k} a_{ik} q_i$$
$$Q_k = \sum_{i \in S_k} a_{ik} p_i + \sum_{i \in T_k} a_{ik} p_i \qquad (2.4)$$

$$\text{where } S_k = \{i : a_{ik} > 0\}$$
$$T_k = \{i : a_{ik} > 0\}$$
$$p_i \le v_i \le q_i$$

The upper and lower bounds on the dual variables are initially determined by the primal constraint types. See figure 2.

Primal (max)	Variables ≥ 0	Variables free	Constraints =	Constraints $\leq$
Dual (min)	Constraints $\geq$	Constraints =	Variables free	Variables ≥ 0

Primal (min)	Variables ≥ 0	Variables free	Constraints =	Constraints $\geq$
Dual (max)	Constraints $\leq$	Constraints =	Variables free	Variables ≥ 0

Figure 2 *Relationship between primal/dual variables and constraints*

If the lower cost P_k is less than the primal cost coefficient (dual right hand side value) c_k then $\sum_{i=1}^{m} a_{ik} v_i \geq P_k > C_k$. This together with the dual constraint $\sum_{i=1}^{m} a_{ik} v_i + y_k - w_k = c_k$ implies that y_k must be zero and w_k must be positive in the optimum solution.
By the rules of complementary slackness we have

$$\begin{aligned} y_k (x_k - u_k) &= 0 \\ w_k (x_k - l_k) &= 0 \end{aligned} \qquad (2.5)$$

As w_k is positive, the corresponding primal constraint must be binding, ie. $x_k = l_k$ therefore the the variable may be fixed to its lower bound.

If the upper cost Q_k is less than c_k then $\sum_{i=1}^{m} a_{ik} v_i \leq Q_k < c_k$. This implies that y_k must be positive and w_k is zero in the optimal solution. Again from equations (2.5) the variable may be fixed at its upper bound, ie. $x_k = u_k$

Figure 1 in Appendix A contains the pseudocode for this subroutine.
If a variable is to be fixed at a bound that is infinite then this implies unboundedness which is detected by the program and so the program halts. This is not detailed in Figure 2 of the Appendix A.

2.2 SNGCOL - 'replacing' singleton columns with bounds on shadow prices

The subroutine SNGCOL detects singleton columns. A singleton column in the primal problem is a column of the A matrix with just one non-zero coefficient. Assume that a_{ik} is the non-zero coefficient of a singleton column k and that x_t has lower and upper bounds l_k and u_k where $l_k \neq u_k$ ($l_k=u_k$ is not considered as in this case x_k would be fixed). The subroutine aims to fix the variable x_k (if possible) at one of its bounds. Suppose that $x_k<u_k$ and if a contradiction occurs then x_k must be equal to its upper bound. Then in the dual problem, $y_k=0$ and $w_k \geq 0$ from (2.5). Thus the dual constraint is $a_k v_k = c_k + w_k$.

If $a_{lk} < 0$ then $v_l \geq \frac{c_k}{a_{lk}}$ (since $w_k>0$). This gives a lower bound on v_1, and if this is greater than the existing lower bound p_1 then the lower bound is tightened. If, however, this is greater than q_1 there is a contradiction and so x_k is fixed to its upper bound.

If $a_{lk} < 0$ then $v_l \leq \frac{c_k}{a_{lk}}$ (since $w_k> 0$). This gives an upper bound on v_1 and if this is less than the existing upper bound q_1 then the upper bound is tightened. If, however, this is less than p_1 there is a contradiction and so x_k is fixed to its upper bound.

An alternative algorithm would be to attempt (where possible) to fix x_k at its lower bound. In this case the opposite assumption, namely $l_k<x_k$ is made (ie. $y_k \geq 0$, $w_k=0$) and a similar argument follows as above. If $a_{lk}>0$ then q_1 may be tightened or x_k may be fixed to its lower bound and if $a_{lk} < 0$ then pi may be tightened or x_k may be fixed to- its lower bound. To use both algorithms as suggested by [BRMIWI,1975] would involve storing two sets of p_1 and q_1 since if p_1 was tightened by the first algorithm this new p_1 could not be used in the second algorithm. This seemed unduly complicated in practice so our implementation handles the first (upper bound) algorithm.

Singleton columns are detected but are not removed. This is because there may be another singleton column with a non-zero entry in row 1. The algorithm may tighten p_1 or p_1 again for another column $j \neq k$. Then on the next pass these new bounds may be used to fix the variable at its upper bound.

Our current implementation calculates $\frac{c_k}{a_{lk}}$ on each pass. An alternative method may be to store these values together with the indices l and k to avoid duplicating these calculations. In addition if for a singleton column with its non-zero entry in row 1 $p_1 > p_1$ then all singleton columns with non-zero coefficients in row 1 may also be fixed. We have chosen not to implement this because this assumes that the A matrix and the objective coefficients remain unchanged. As discussed later further procedures, such as the identification and reduction of doubleton rows, may be added to our code which do alter these values. Therefore we chose a more flexible approach in our implementation.

Figure 2 in Appendix A specifies the pseudocode for this second subroutine, **SNGCOL**.

2.3 REDROW - detecting redundant rows

This subroutine detects redundant rows in the primal problem by analysing the lower and upper constraint bounds and sets these redundant rows free. Lower and upper bounds, L_i and U_i, on the i^{th} constraint are calculated by taking into account the variable bounds as follows.

$$L_i = \sum_{j \in P_i} a_{ij} l_j + \sum_{j \in N_i} a_{ij} u_j$$

$$U_i = \sum_{j \in P_i} a_{ij} u_j + \sum_{j \in N_i} a_{ij} l_j$$

$$where\ P_i\ \{j: a_{ij} > 0\}, \quad N_i = \{j: a_{ij} < 0\}$$

If $U_i \leq b_i$ the constraint is redundant and so is set free. This also implies that the corresponding dual variable v_i can be fixed to its lower bound, i.e. q_i is set equal to p_i. If $L_i > b_i$ then the constraint i can not be satisfied and the problem is declared infeasible. Furthermore, if $L_i = b_i$ then row i is redundant and all x_j, $j \in P_i$ are fixed at $x_j = l_j$ and all x_j, $j \in N_j$ are fixed at $x_j = u_j$.

The pseudocode for this subroutine is provided in Appendix A (figure 3), As mentioned previously, this only details the pseudocode for the case where the primal constraints are as shown in (2.1). Other cases are considered in [BRMIWI,1975].

2.4 REMBND - tightening variable bounds

The subroutine REMBND removes or tightens the primal variable bounds. This subroutine is not performed in the first pass as it is necessary to compute variable bounds by considering the bounds on the constraints which are not determined until the subroutine REDROW is executed.

New variable bounds are constructed by examining each constraint. Let us consider the i^{th} constraint, then for the k^{th} variable we have

$x_k \leq \frac{1}{a_{ik}} \{b_i - \sum_{j \neq k} a_{ij} x_j\}$ assuming that $a_{ik} > 0$ and since $\sum a_{ij} x_j = L_i - a_{ik} l_k$,

it follows that $x_k \leq \frac{1}{a_{ik}} \{b_i - L_i\} + l_k$

Similarly, if $a_{ik} < 0$ a new lower bound for x_k, $x_k \geq \frac{1}{a_{ik}} \{b_i - L_i\} + u_k$ is obtained.

If these new bounds are tighter than the existing ones the variable bounds are updated. If a new lower bound is computed which is greater than the upper bound (or a new upper bound is found which is less than the lower bound) infeasibility is detected and the program stops. Furthermore, if a new lower/upper bound is calculated to be equal to the upper/lower bound the variable is fixed at this bound.

It may be argued that a redundant variable bound should be simply removed rather than tightened as this may result in acceleration in the simplex algorithm as there will be less bounded variables. However, it is preferable to obtain a tighter formulation of the problem especially for discrete programming problems discussed later and for this reason variables are not set free. In addition this avoids the spurious unbounded condition which may occur from freeing variables, as discussed by Tomlin and Welch [TOMWEL, 1983a].

The pseudocode for subroutine REMBND is provided in the Appendix A (figure 4).

2.5 SNGROW - replacing singleton rows by simple bounds

The final subroutine, SNGROW is the dual case of SNGCOL. Here singleton rows are replaced by simple bounds.

A singleton row i such as $a_{ik}\, x_k \leq b_i$ implies new bounds for the variable x_k.
That is,

$$x_k \leq \frac{b_i}{a_{ik}} \qquad if\ a_{ik} > 0 \qquad or$$

$$x_k \geq \frac{b_i}{a_{ik}} \qquad if\ a_{ik} < 0$$

Thus, if these bounds are tighter than the existing variable bounds, u_k and 1_k then the variable bounds are updated and the singleton row is set free. If the new bounds conflict with the old ones, infeasibility is detected and the program terminates.

The pseudocode for this subroutine is provided in the Appendix A (figure 5).

3. Presolve Data Structure & Communication with FortLP

The Presolve algorithm is included, as an option, in FortLP's optimization framework. FortLP's data structure consists of packed columns but, since many of the Presolve subroutines involve row-wise operations, a new data structure is used which enables both column and row scanning. Thus, the Presolve data structure comprises linked lists for both rows and columns. Each list has a header which refers to the first non-zero elements of each row or column, so the i-th element of the column\row header refers to the first element of the i-th column\row. Other arrays contain the right hand side (rhs) values, column and row types, primal and dual variable bounds.

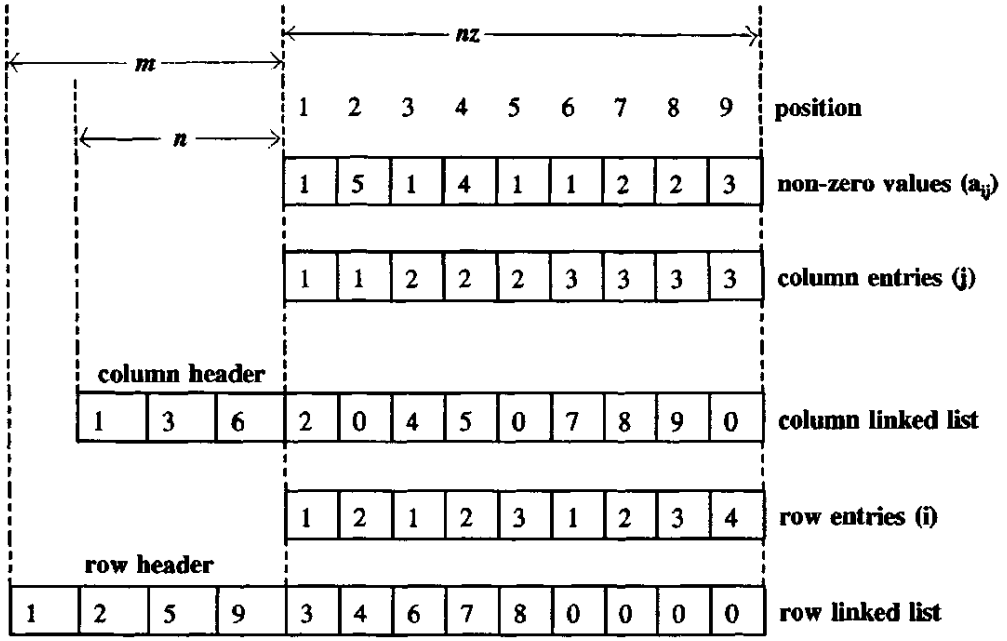
The A matrix is stored column-wise and row-wise. The first array stores the non-zero values (in column order); the second array holds the column indices for these non-zero values; the third array is a column linked list with an additional column header containing the locations of the first elements in the columns; the fourth array holds the row indices for these non-zero values and the fifth array is a row linked list with an additional row header locating the first non-zero entries in the rows.

The example below illustrates the data structure used to store the given A matrix.

Example *Presolve Data Structure*

$$\begin{pmatrix} 1 & 1 & 1 \\ 5 & 4 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 3 \end{pmatrix}$$

number of row(m) = 4
number of columns(n) = 3
number of non – zeross(nz) = 9



From the column header, the first non-zero entry in column 1 is located in position 1 of the column link array. In fact all information relating to this entry is found in position 1 of all the arrays. Thus its value is in position 1 of the non-zero array. The column and row entry arrays indicates that this non-zero entry is in column 1 and row 1. The column link array has a 2 in position 1 indicating that the next non-zero entry in this column is located in position 2. Hence position 2 of all the arrays provides information about this entry. If a column (or row) link array has a zero entry then there are no more entries in the column (or row), so the zero in position 2 of the column link array marks the end of column 1.

The row linked list together with its row header stores the A matrix in a similar way but row-wise, so, for example, from the row header the first entry in row 3 is located in position 5. This has value 1 (from position 5 of the non-zero array), is in column 2 (from the column entry array) and the next element in this row is stored in position 8 (from the row link list).

The Preserve program contains subroutines (Inipre and Outpre) that provide communication with the FortLP optimizer. The first of these initializes the Presolve data structure and copies the upper bound values of the variables, the rhs values, the A matrix coefficients (including the objective row), the row and column types from FortLP.

As Presolve occurs after set-up, the lower bounds on the variables are zero. Upon completion of the Presolve subroutine, the updated information is communicated back to FortLP so in order to maintain the set-up conditions the variable bounds must be translated so as to restore the lower bounds to zero. Thus, the upper bound value for each variable communicated back to FortLP is Presolve's upper bound value minus the lower bound value. In addition, the FortLP array (RLOFXV), storing the original lower bounds before set-up, is^updated with the new lower bound from Presolve.

4. Results

Table 1 below provides the results from applying Presolve to various problems including some netlib problems. It provides the number of redundancies detected, variable fixed and variable bounds tightened together with the problem statistics.

It is anticipated that further improvement may be obtained by the addition of other reduction procedures, in particular the elimination of doubleton rows (see section 5).

Further results are needed which report the time and number of iterations required to solve these models with and without Presolve. However, as the communication with FortLP was not complete at the time of writing and as FortLP has now been replaced by FortMP, these results were not complete. Once Presolve is incorporated into the new FortMP, these tests will be performed.

TABLE 1

Problem Name	Rows	Columns	No. of non-zeros	No. of Redundant Rows	No. of Fixed columns	No. of bounds tightened	Rows (reduced model)	Columns (reduced model
afiro	28	32	88	4	0	32	24	32
25fv47	821	1571	11127	43	27	1254	778	1544
ganges	1310	1681	7021	185	184	1032	1225	1497
8800	261	87	1000	99	9	60	162	72
gray2	35	48	144	0	0	24	35	48
gray9	63	96	288	0	0	48	63	96
bsc	439	209	1604	160	0	95	279	209
egout	99	144	392	24	24	55	75	117
modglob	292	422	1390	2	33	289	290	389
brandy	221	249	2150	70	44	101	221	151

Tightening bounds on variables may result in setting bounds on variables which were originally free. This may cause the simplex algorithm to take longer, so further investigation is required and it may be preferable not to communicate these tightened bounds back to FortLP. For example, our preliminary tests (not reported here) showed that for a problem such as gray2, where the only reduction performed was the tightening of bounds, the simplex algorithm took longer with the reduced model. In addition, the tightening of a bound may suggest that the variable can be freed. For example, if a variable originally has the conventional bounds $[0,\infty)$ and at the end of the reduction procedure has the new bounds $[5, \infty)$ say, then the variable will always be greater than the original lower bound so it may be set free.

5. Post Processing and Scope for Future Work

5.1 The need for post processing

It is well known that applying the simplex algorithm to a reduced problem may result in a solution that is not formally optimal [TOMWEL, 1983b]. That is, the objective function has the correct optimal value, but the basis is incorrect. This occurs when the problem has a degenerate optimum solution, ie. there is a redundant constraint which is binding at the optimum solution, so the optimum basis is not unique. Thus solving the reduced model may result in a solution with an alternate basis to that obtained by solving the original problem and so there may be different dual values present. If dual optimality is required (for example for post optimal analysis) then the dual simplex algorithm may be applied to obtain the 'formally' optimal solution. The solution to the dual problem in this case will have alternate basic solutions so cycling will occur which is computationally costly. Another cause for this lack of 'formally' optimum solution may occur when a bounded variable is non-basic in the optimum solution of the original problem and the bounds of this variable are tightened by the reduction procedure. Then in the reduced problem this variable will be non-basic but at a new upper or lower bound. This means that the solution to the reduced problem is optimum but not basic. To overcome this it is necessary to take the optimal basis of the reduced problem, restore the bounds on the variables and perform invert. The variables which are at their new tightened bounds will give primal infeasibility with the original problem and so it is will be necessary to continue using dual simplex to obtain the formally optimal solution. This too may be very computationally costly.

Tomlin and Welch propose an alternative method for obtaining a, formally optimal solution by reconstructing the original constraints and variables (Ibid.). Currently, our code does not include a postsolve procedure. Clearly there is a need for such an addition but this requires further study.

5.2 Other reduction procedures

Alternative procedures exist for reducing LP problems. For an extensive review and discussion of such procedures see (KALITEZI,1983). In addition Williams [WILLIAMS, 1982] proposes a revised procedure for implementing the algorithm provided by [BRMIWI, 1975] which consists of two phases. In the first phase bounds on variables are tightened and bounds on shadow prices are relaxed while in the second phase bounds on variables are removed (where possible) and bounds on shadow prices tightened. The procedure involves the scanning of columns which may result in the tightening of variable bounds, the fixing of variables and the detection of singleton columns and at the end of each pass rows are scanned which may detect redundancies, infeasibility or singleton rows. Unlike our algorithm, when a singleton column is detected it is *replaced* by shadow price bounds. This subroutine does not fix variables as in our algorithm. A further subroutine is included in Williams' procedure which is the dual of REMBND. This may tighten or remove bounds on the dual variables. This procedure involves the freeing of variables in phase two which may lead to a spurious unbounded condition as mentioned earlier and as discussed by [TOMWEL, 1983a].

More recently there have been other developments. For example, the elimination of duplicate rows [TOMWEL,1986]; the nullification of balancing constraints [ANDBAR,1992]; elimination of doubleton rows and rows with all but one element of the same sign [IBM, 1992].

The detection of duplicate rows [TOMWEL,1986] involves identifying constraints which are identical except for a scalar multiple. This is carried out by making a single pass of the matrix, scanning the non-fixed columns. Rows are partitioned into potential duplicates by assigning a scalar factor to each row and dividing all elements in the row by this factor. At the end of the pass, the duplicate rows in the A matrix are found, redundancies eliminated and infeasibility detected. This algorithm is not costly computationally as it only involves one pass of the matrix, but the amount of reduction achieved by this procedure is more limited than those detailed in our Presolve algorithm. It could be argued that the existence of duplicate rows in a model are a result of bad formulation. However, detecting such redundancy prior to optimization provides some defence. The same is true for the detection of infeasibility or unboundedness.

The nullification of balancing constraints algorithm [ANDBAR, 1992] implemented by Andre and Barbulescu is concerned with performing linear transformation on the coefficient matrix to globally eliminate continuity constraints. Such a process may also increase the density of the matrix, yet dramatically reduce the number of iterations in the simplex algorithm (op. cit. p22). Andre and Barbulescu propose a method for avoiding an increase in the CPU time by using a stop criterion defined by Knolmayer [KNOLMA, 1982]. Such a measure can determine whether the total CPU time will be increased if the presolution is continued and if an increase is predicted the presolution process may be halted and the optimization commenced.

The elimination of doubleton rows involves identifying equality constraints with exactly two non-zero coefficients, substituting for one of these non-zero elements and setting the row free. For example, if $a_{ij}x_j + a_{ik}x_k = b_i$ is a doubleton row then x_j is substituted with $\frac{b_i}{a_{ij}} - \frac{a_{ik}}{a_{ij}}x_k$

In performing this substitution the coefficients in the A matrix must be altered. This involves scanning the coefficient matrix row by row and locating constraints with the substitution element. When such a row is found the other doubleton coefficient must be obtained and updated. This may result in increasing the number of non-zeroes in a row. Nevertheless, results from OSL's preprocessing routine EKKPRSL (see Appendix B) indicate that this type of reduction is most beneficial. However, as such a procedure may increase the density of the matrix and requires a simultaneous row scan and double column scan it has greater complexity. Therefore the CPU time of the Presolve algorithm is increased, but in addition the CPU time of the optimizer is decreased.

The elimination of equality rows with all but one element of the same sign, for example

$$a_1x_1 - a_2x_2 - \cdots - a_nx_n = b$$

is a similar to the doubleton row elimination algorithm though involving multiple column and row scans. In this case the complexity in implementing this procedure is greater than that of the doubleton row. These two algorithms require more computational effort that the Presolve algorithm implemented in FortLP (described earlier), but produce greater levels of reduction.

5.3 Implementation considerations for the doubleton reduction procedure

The doubleton row reduction procedure consists of the following steps:

(i) identify a doubleton row and flag it
(ii) substitute for the first variable
(iii) store doubleton column links

The first step is easy to implement with our existing data structure and subroutines GETFRW (get the first non-zero element in the row) and GETNRW (get the next element in the row): for each row simply perform GETFRW followed by GETNRW to obtain the second non-zero element in the row. If this is the last element then it is a doubleton row and so flag it. It is preferable to define a new row type to identify doubleton rows rather than just freeing them as this makes it easier to restore the eliminated variables after optimization.

The second step involves a simultaneous column scan for the two variables in the doubleton row. For example, assume there is a doubleton row in row i with non-zero entries in columns j and k. Then for each non-zero element in column j (ie. $a_{lj} 1 \neq i$) the 1^{th} element in column k must be updated. If the corresponding element a_{ik} is non-zero then it is replaced by $a_{ik} - a_{lj} \frac{a_{ik}}{a_{ij}}$.

If however, it is zero then a new entry $-a_{lj} \frac{a_{ik}}{a_{ij}}$ must be added to the data structure.

In addition the right hand side value b, must be similarly replaced by $b_l - a_{lj} \frac{b_i}{a_{ij}}$.

These substitutions are also performed in the objective row. Adding a new entry to the A matrix is easily carried out in our data structure by adding the new entry at the end of the arrays and updating the linked lists. This however must be communicated back to FortLP. The addition of new non-zero elements to the right hand side vector may be necessary as a result of existing procedures in our algorithm. This is communicated back to FortLP by allowing extra storage in the necessary arrays. A greater allowance will be needed to account for the additions created by this substitutions.

The third step is required so that the eliminated variables may be reinstated after optimization by re-substitution. For each doubleton row, the column entries, j and k are stored so that a record is kept of which variable has been substituted. Then after optimization it is easy to scan this information and obtain values for the missing variables by back substitution. Obviously, this information must also be communicated and stored within FortLP's data structure.

6. Final Comments

Our implementation of the presolve algorithm detailed in [BRMIWI, 1975] performs considerable reduction on the test problems. It is anticipated that dramatic speed up of solution will be achieved once this is fully incorporated into the new optimizing system FortMP (which replaces FortLP). The elimination of doubleton rows may later be implemented to achieve further reduction and subsequent acceleration in solution. The merit of all reduction procedures depend on the balance between the reduction obtained and the time or cost expended in achieving this reduction. Providing that the combined CPU time of the presolver, the solver (with the reduced model) and the postsolver (if used) does not exceed the CPU time of the solver (with the original problem), the reduction algorithm is worthwhile.

The new FortMP handles ranges on constraints. These may be dealt with by adding bounds on slack (or surplus) variables corresponding to the appropriate row. These can be simply read in together with the model and no further changes will be required to the Presolve algorithm.

A Postsolve procedure is necessary for obtaining 'formally' optimal solutions. This is particularly important when meaningful dual values are required, for example for post-optimal analysis. The techniques used in basis recovery in IPM may be used in implementing such a procedure.

The presolve procedure may be extended to be used in Discrete Programming, in particular at each stage of the Branch and Bound (B & B) algorithm since at each node of the B & B tree an LP problem must be solved. However, in order to deal with integer variables some modifications are needed. In the subroutines where new bounds are computed, these bounds must be rounded to the appropriate integer value (ie. the new lower bound l_k is rounded to (l_k+ 1-$\in$) and the upper bound u_k is rounded to (u_k+$\in$) where e is a small number). Also the replacement of a singleton column by a shadow price bound is not valid for integer variables unless the non-zero coefficient in the singleton column is in a row where all non-zero entries correspond to integer variables and the coefficient is a divisor of all these non-zero entries as well as the right hand side value [WILLIAMS, 1982]. The tightening of bounds of variables is more important for integer problems and so unlike our suggestion in section 4 all new bounds should be communicated to the optimizer and variables should not be freed.

REFERENCES

[ANDBAR,1992] Andrei, N. & M. Barbulescu (1992)
Balance Constraints Reduction of Large-Scale Linear Programming Problems, Research Institute for Informatics, Analysis & Mathematical Modeling of Systems, Bucharest, Romania

[BRMIWI,1975] Brearley, A.L., G. Mitra & H.P. Williams (1975)
Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm, Mathematical Programming 8 (1975) pp54-83, North-Holland

[IBM, 1992] *Optimization Subroutine Library (1992), Guide and Reference IBM Corporation*

[KALITEZI,1983] Karwan, M.H., V. Lotfi, J. Telgen & S. Zionts (eds)
Redundancy in Mathematical Programming, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag

[KNOLMA, 1982] Knolmayer, G.F. (1982),
Computational experiments in the formulation of linear product-mix and non-convex production-investment models, Computing & Operations Research, Vol. 9 No. 3 pp207-219

[LEVKOV,1992] Levkovitz, R. (1992),
An Investigation of Interior Point Methods for Large Scale Linear Programs: Theory and Computational Algorithms
PhD Thesis, 1992, Department of Mathematics & Statistics, Brunei University

[TOMWEL,1983a] Tomlin, J. & J.S.Welch (1983),
A Pathological Case in the Reduction of Linear Programs, Operations Research Letters, Vol.2, No.2, pp53-57

[TOMWEL,1983b] Tomlin, J. & J.S.Welch (1983),
Formal Optimization of Some Reduced Linear Programming Problems, Mathematical Programming 27 (1983) pp232-240, North Holland

[TOMWEL, 1986] Tomlin, J. & J.S.Welch (1986),
Finding Duplicate Rows in a Linear Programming Model, Operations Research Letters Vol.5, No.l, pp7-ll

[WILLIAMS, 1982] Williams, H.P. (1982),
A Reduction Procedure for Linear and Integer Programming Models in "A comparative Study of Methods for Identifying Redundant Constraints in Linear Programming", edited by J. Telgen & s. Zionts, Springer-Verlag

[WILLIAMS, 1990] Williams, H.P. (1990),
Model Building in Mathematical Programming, Third Edition, John Wiley

APPENDIX A

```
begin
  if j is a singleton column then
        if a_ij > 0 then

              newp = c_j / a_1j          {Calculate new lower bound
                                          for shadow price}
              if newp > p_1 then
                p_1 = newp               {Tighten bound}
              endif
              if newp > q_1 then
                x_i = u_i                {Fix variable to its upper bound }
                update rhs values
                reduct = 1
              endif
        else
              newq = c_j / a_1j          {Calculate new upper bound
                                          for shadow price }
              if newq < q_1 then
                q_1 = newq               {Tighten bound}
              endif
              ifnewq , p_1 then
              X=|                        {Fix variable to its lower bound}
                update rhs values
                reduct = 1
              endif
            endif
        endif
```

Figure 2 *Subroutine SNGCOL*

```
Begin
 For j=l to n do
  If X_j is not fixed then
        Initialize bounds P_j = Q_j = 0
        While there is a non-zero element  a_ij  in column j do
              Get a_ij
              If row i is a non-redundant constraint then
                    If  a_ij  >0 then
                      P_j = P_j  +  a_ij p_i                {calculate P_j and Q_j}
                      Q_j = Q_j  + a_ij q_i
                    else if  a_ij  <0 then
                      P_j = P_j  +  a_ij q_i
                      Q_j = Q_j  + a_ij p_i
                    endif
               endif
              enddo
              If  P_j > c_j  then                {x_j is fixed at its lower bound}
                Fix x_j = l_j
                reduct = 1
              else if Q_j < C_j then       {X_j is fixed at its upper bound} Fix x_j = u_j
                    reduct = 1
              endif
         endif
        enddo
end
```

Figure 1 *Subroutine FIXCOL*

```
begin
  Initialize L(i)=U(i)=0 ∀i=1..m
    for i=l to m do
      If row i is a non-redundant constraint then
        while there is a non-zero element a_ij in row i do
          If column j is not fixed then
            if a_ij > 0 then
              L_i = L_i + a_ij l_j
              U_i = U_s + a_ij u_j
            else if a_ij < 0
              L_i = L_i + a_ij u_j
              U_i = U_i + a_ij lj
            endif
          endif
        endo
        If U_i ≤ b_i then
          free constraint i                    {constraint i is redundant}
          q_i= p_i                             {Fix dual variable to lower bound}
          reduct = 1
       else if L_i > b_i then
          Problem is infeasible → EXIT
       else if L_i = b_i then
        for j=l to n do
          if x_j is not fixed then
            if a_ij > 0 then
              x_j = l_j
            else
              x_j = u_j
            end if
            update rhs values
            reduct = 1
          end if
       enddo
       endif
  enddo
end
```

Figure 3 *Subroutine REDROW*

```
begin
    for  j = 1 to n do
       for  i = l to m do
         if a_ij > 0 then
            newu = l_j + (1/a_ij)(b_j − L_j)
         else if a_ij < 0 then
            newl = l_j + (1/a_ij)(b_i − U_i)
         endif
       endo
       if (newl > u_k) or (newu < l_k) then
         The problem is infeasible → exit
         endif
       if   newl > l_k then
            l_k = newl
      endif
       if   newu < u_k then
            u_k = newu
        endif
         if u_k = l_k then
            x_k = u_k
            update rhs values
            reduct = 1
         endif
     endo
end
```

Figure 4 *Subroutine REMBND*

```
begin
    if i is a singleton row then
      if a_ik > 0 then

        newu = b_i / a_ik
        if newu < l_k then
          The problem is infeasible → EXIT
        endif
        if newu < u_k  then
          u_k = newu
        endif
      else if a_ik < 0 then
        newl = b_i / a_ik
        if newl > u_k  then
          The problem is infeasible → EXIT
        endif
        if newl > l_k then
          l_k = newl
        endif
    endif
 endif
end
```

Figure 5 *Subroutine SNGROW*

```
Initialize
while pass<2  do
     call    fixcol
     call    sngcol
     If  tpass  ≠ 1  then
        call rembnd
     endif
     call   redrew
     call   sngrow
     If  reduct  =  0  then
        pass = pass +1
     else
        pass = 0
     end if
     tpass = tpass  +1
enddo
```

Figure 6 *Presolve Main program*

Results from EKKPRSL (OSL's reduction subroutine)

Problem	Original Model		Level 1 Reduced		Level 2 Reduced		Level 3 Reduced	
	Rows	Columns	Rows	Columns	Rows	Columns	Rows	Columns
25fv47	821	1571	724	1493	759	1528	715	1484
80bau3b			2049	9248	2049	9248	2049	9248
afiro	28	32	21	30	25	32	21	30
ganges	1310	1681	787	1169	835	1208	614	996
brandy	221	249	118	200	118	200	110	192
cook	123	133	115	133	115	133	115	133
d2q06c	2171	5167	1996	5055	2026	5082	1937	4993
degen3	1504	1818	1412	1727	1503	1818	1412	1757
israel	174	142	163	142	163	142	163	142

Level 1 reduction includes the elimination of doubleton rows
Level 2 reduction includes the elimination of rows will all but one element of the same sign (but not doubleton row elimination)
Level 3 reduction includes everything.